

# Browser-based distributed evolutionary computation: performance and scaling behavior

Juan-J. Merelo  
 Antonio Mora-Garcia  
 JLJ Laredo  
 Dpto. Arquitectura y  
 Tecnología de Computadores  
 ETS Ingeniería Informática  
 Universidad de Granada,  
 Granada 18071  
 {jmerelo|amorag|juanlu}@geneura.ugr.es

Juan Lupion  
 The Cocktail  
 pantulis@gmail.com

Fernando Tricas  
 Dpto. Informática e Ingeniería  
 de Sistemas  
 Centro Politécnico Superior  
 Universidad de Zaragoza  
 ftricas@unizar.es

## ABSTRACT

The challenge of ad-hoc computing is to find the way of taking advantage of spare cycles in an efficient way that takes into account all capabilities of the devices and interconnections available to them. In this paper we explore distributed evolutionary computation based on the Ruby on Rails framework, which overlays a Model-View-Controller on evolutionary computation. It allows anybody with a web browser (that is, mostly everybody connected to the Internet) to participate in an evolutionary computation experiment. Using a straightforward farming model, we consider different factors, such as the size of the population used. We are mostly interested in how they impact on performance, but also the scaling behavior when a non-trivial number of computers is applied to the problem. Experiments show the impact of different packet sizes on performance, as well as a quite limited scaling behavior, due to the characteristics of the server. Several solutions for that problem are proposed.

## Keywords

Distributed computing, internet computing, world-wide-web, overlay networks, application level networks, ruby on rails, parallel computing, implementations

## 1. INTRODUCTION

Application-level networks, ALNs, are configured as a set of clients that can provide their spare CPU cycles by means of an application that can be downloaded, establishing a distributed computation network. Some ALN like SETI@Home have been quite successful, while other experiments such as Popular Power have not. Many of these ALNs provide spare or ad hoc computational power for distributed computing experiments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

The key feature of these application-level networks is the simplicity of use: we believe that the best way to obtain the participation of as many users as possible is to avoid trouble. In particular, it will be easier if they do not need to download a special application to participate. For this reason, we are exploring the use of elements that are usually installed in the user's computer; in this sense, it is clear that the web browser is an element almost universally installed: it is available even in some cellular phones. Moreover, most browsers include a JavaScript interpreter [18, 32, 15]. JavaScript is an interpreted language, initially proposed by Netscape, and later adopted as an ECMA standard [11, 12, 13, 14]. In this way, most browsers are compatible, at least at a language level (not always at the level of browser objects, where there exists a reasonable compatibility, anyway).

The ability to use these features for distributed computing appeared with the `XMLHttpRequest` object, which allows asynchronous petitions to the server, in what has been called AJAX, Asynchronous JavaScript and XML [36]<sup>1</sup>. The traditional client/server model becomes then more egalitarian, or closer to a peer to peer model, since a bidirectional communication line appears: the browser can make calls to the server, do some computation and later send the results to the server. The proposed mechanism is as follows: the `XMLHttpRequest` is provided with a request to the server and a pointer to a *callback* function. The request generates an event, which is asynchronously activated when a reply is received making use of the *callback* function. Following this approach the browser is not locked, providing the way to program applications that are similar to the ones the volunteers are used to, in the sense that they do not have to wait for the application to load and render the whole screen every time a request is made. On the other side, this provides a way to use the browser for application level networks and its use for distributed computing systems, since the request-response loop does not need the user participation in a fashion very similar to any other distributed computing application. This feature can be controlled from the server with any programming language. Of course, it can also be combined

<sup>1</sup>AJAX is just one of the possible ways to perform asynchronous client-server communication, the others being AJAJ (Asynchronous Javascript and JSON), and *remoting* using applets or embedded objects. However, it is quite popular, and a wide user base and documentation is available for it.

with other distributed programming frameworks based on OpenGrid [28].

The server can be programmed traditionally using any of the paradigms available (servlets or CGIs, for instance), but in order to produce a rapid development of the application, the use of Ruby on Rails [33, 21, 34] was considered [23]. It is a framework based on Ruby language and in the Model/View/Controller [16, 20] paradigm (which has been used before in evolutive computing; for example, in [5]). In this context the data model is clearly separated (usually with a database management system) from the different *views* (HTML templates that will be used by the server to fill in the data and send the pages to the client), and from the control part, the functions that modify and manage data; in RoR, the controllers are in charge of receiving client requests and react to them. When constructing a RoR application we need to setup a data model, a controller which will distribute the computation among client and server and a set of views. In our case, these views will include the client's computation part (since the JavaScript programs are included in the pages that will be served to clients)<sup>2</sup>.

We are concentrating on distributing, evolutionary computation applications, which has already been adapted to several paradigms of parallel and distributed computing (for example, Jini [17], JavaSpaces [31], Java with applets [9], MPI [8, 7], service oriented architectures [25, 27] and P2P [3, 26]) and it is adequate for this kind of exercise for several reasons: it is a population based method, so computation can be distributed among nodes in many different ways; besides, some works suggest that there are synergies among evolutive algorithms and parallelization: isolated populations that are connected only eventually avoid the lost of diversity and produce better solutions in fewer time obtaining, in some cases, superlinear accelerations [1].

This work is more than a proof of concept [23]: first, we will try to establish a baseline for the performance of a JavaScript-based evolutionary algorithm by running benchmarks on several virtual machines; then, we will try to see how different elements of the system, especially latency, influence performance, and, finally, we will do some measurements of DCoR system (*Distributed Computation on Rails*) in real network, in order to see how this computation scales.

The rest of the paper is organized as follows: next, an exposition of the state of the art in volunteer and so-called *parasitic* computing is presented. Section 3 presents the DCoR (Distributed Computation on Rails) system; to be followed by experiments in browser performance (section 4) and scaling behavior (section 5). Finally, the last section will present conclusions and future lines of research.

## 2. STATE OF THE ART

So called *volunteer computing* [29, 30, 2] takes advantage of the creation of an infrastructure so that different people can donate CPU cycles for a joint computing effort. The best known project is SETI@home<sup>3</sup>, which, from the user's point of view is a screen-saver which has to be downloaded and installed; when the user's CPU is not busy it performs

<sup>2</sup>Please note that this is only one possible arrangement, which was deemed the simplest for these initial studies. Others, including storing JS code in the database (making it part of the model) are also possible within RoR

<sup>3</sup>See <http://setiathome.berkeley.edu/> for downloading the software and some reports.

several signal analysis operations. Some companies related to volunteer computing, such as Popular Power (and others; they are referenced, for example, in [6]) did some experimentation with Java based clients, but none has had commercial success.

There are mainly two problems in this kind of networks: first of all, it is important not to abuse volunteers CPU resources; secondly, a sufficient number of users is needed in order to be able to do the required computation; this can also be a problem on its own if there are too many users for the considered setup. A third problem is that performance prediction is difficult when neither the number of participants nor their individual node performances are known in advance. In any case, we believe that the best way to obtain enough users is to make it easy for them to participate, using technologies available in their computers, as the browser is. In fact, some suggestions have been published (for example, the one of Jim Culbert in his weblog [10], and in some mailing lists), but we are not aware of any serious study about it.

The proposed approach could also be considered as *parasitic computing* since, as stated in Section 1, the only participation of the user will be to load a web page; in fact, it could use these resources without his acquiescence (and, in any case, it would be desirable to run without causing much trouble). The concept was introduced by Barabási in [4], and followed by others (for instance, Kohring in [19]). In that work they proposed to use the Internet routers to compute a *checksum* by means of a set of specially crafted packets to solve the SAT problem. Anyway, although the concept is interesting, there seems not to be a continuation for this work.

The virtual machine embedded into the browser provides a way to easily do that kind of sneaky/parasitic computing, but JavaScript has the problem to be an interpreted language and the efficiency of different implementations varies wildly. Moreover, it is not optimized for numerical computation but for object tree management (the so called DOM, document object model) and strings. Nevertheless its wide availability makes us think about considering it, at least as a possibility. It is also important to remember that these resources can be used without the user's participation (they only need to visit a web-page) opens a wide set of alternative possibilities (and dangers, of course).

In this work an evolutive computation system will be presented. It has been developed in a Ruby on Rails based framework that takes advantage of this feature; in this sense the approaches is new. We will tackle the three main problems: abundance of clients (via the system itself), possibility of client CPU abuse (also via the system itself; the JS virtual machine runs within a sandbox inside the browser), and performance prediction (which we will try to approach via several experiments and benchmarks).

## 3. RESOURCES AND METHODOLOGY

For the experiment we need several clients with JavaScript equipped browsers and a server running Ruby on Rails. RoR applications include their own web server, WEBrick, but there are other options, such as Mongrel<sup>4</sup> and lighttpd<sup>5</sup> that are faster and will be preferred for the experiments described

<sup>4</sup><http://mongrel.rubyforge.org/>

<sup>5</sup><http://lighttpd.net/>

below.

The application follows MVC model; in this sense it is organized as a model, a view and a controller. We will concentrate on the first and the last of them.

The **model** is a table in the database representing the population, that could be similar to this<sup>6</sup>:

```
create table guy {
  cromosoma varchar(256),
  fitness float
};
```

This table will store the population. We have preferred a traditional representation using a binary string; the chromosome will be a list of 0s and 1s. We are also using a scalar fitness, represented with a single *float* value<sup>7</sup>. In any case, the data model is related to the application we want to optimize and different chromosomal representations and different fitness would need a different data model.

For the **controller**, we will need controls that request new elements from the population pool and to re-send them once evaluated. This is also related to the labor division among clients and server. We need to take into account that the evolutive algorithm needs to include several actions: evaluation, genetic operators (mutation and crossing), and merging of the population. The simplest way of task distribution is to evaluate fitness parameters (it is usually the most time consuming operation) on the clients and to do the other steps on the server, as shown in the algorithm; this scheme is usually called *farming*. Technically, the evaluation step should be included in the *view*, since it is interpreted in the client; in practice, it will be a JavaScript program that will be included in the templates stored in the corresponding directory of the RoR application. Obviously, the fact that it will be executed on the client has security and authentication consequences that have to be considered (including, probably, fraud as shown in [29]). Since we are doing our experiments in a controlled way, they have not been considered. We will only consider IP-based authentication (that is, in this experiment we know in advance which IP addresses are going to participate in it), and we will suppose that clients will not send a higher fitness than the computed one (which would give false results). Of course, there are other methods to deal with this, such as replicating evaluations in different clients (and comparing them), or using some kind of client/server codification that would hinder or avoid tampering with data. Controls will be needed to generate individuals and for the genetic operators. They will be written in Ruby because they will be executed on the server. The whole system can be sketched as follows, from the client's point of view:

1. Loading of the client code, which will be done along with the web page, identified by an experiment unique URL: it will start when the web page is loaded on the browser (by means of *onLoad* browser's event) or at the user's request.

<sup>6</sup>In fact, it stores also some information about the algorithm it belongs to: an identifier and the state

<sup>7</sup>Which limits us, for the time being, to single-objective optimization; but, in fact, there is no constraint, since current database management systems can work with vectorial data types

2. Request individuals to the server in order to evaluate them. The server sends a prefixed number of individuals (a *package*). If there are not enough individuals to be evaluated, they will be generated on the fly (by applying genetic operators).

3. The client evaluates individuals and send the result back to the server. It will be evaluated in the server by the controller's method *populationReady*. Several formats can be used for the information interchange. Being AJAX the selected technology, it would seem natural to use XML, but we selected JSON (*JavaScript Object Notation*). JSON<sup>8</sup> is an object serialization protocol that uses alphanumeric strings for data structures. It can be evaluated in JavaScript in order to convert it to an object and Ruby can also interpret and produce it from the database in a very straightforward manner.

4. The server uses tournament selection to generate new individuals. In this method from a number  $n$  of individuals, the worst  $p < n$  are suppressed and substituted with the offspring of the rest of individuals; of course it can also be done selecting the best ones until the number of needed individuals is reached (usually the same number as in the original population).

This tournament can be done in several steps by means of random tournaments that will serve to detect the worst individuals and to eliminate them; then we will take the remaining ones in order to reproduce them. The algorithm terminates when a number of individuals has been evaluated or when a prefixed fitness level is reached. In this way a percentage of the new individuals will be generated by means of the available genetic operators.

5. The reply is sent to the client where a *callback* is generated in order to return to the evaluation step with these new individuals.

6. The algorithm terminates when the client stops or when some condition is reached; for example, the prefixed fitness is reached, or the selected number of evaluations has been made.

The parameters and the execution of the algorithm are configured by means of the web page, as shown in Figure 1. The algorithm can be executed by clicking on Run from its web page, where it is also possible to modify the parameters or to restart it, as is shown in the figure mentioned above. In any case, each algorithm has its own URL in a format like <http://node:3000/algorithm/generation/<algorithmID>>, that can be used to run it from any browser. A screen capture of an algorithm running is shown in Figure 2. The web page is dynamically refreshed when new server requests are received. The source code of the project is at RubyForge<sup>9</sup>, and it has been licensed under the GPL; notice that this is an ongoing work and the software state at each moment can or cannot correspond to the ideas expressed in this paper.

## 4. BROWSER PERFORMANCE

Several experiments on different browsers and with a limited number of computers ([24, 23], in Spanish), have yielded

<sup>8</sup>More information in <http://www.json.org/>

<sup>9</sup> <http://rubyforge.org/projects/dconrails/>



Figure 1: Screen capture of the DCoR application running on the browser. It shows different controls and algorithm parameters which will be executed in the tests.

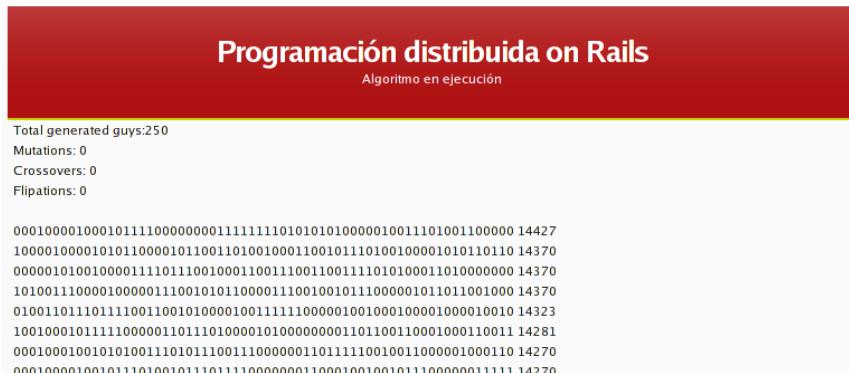


Figure 2: The algorithm is executing. It shows the content of each individual and the fitness value for the 64 bits binary knapsack problem.

the result that different browsers have very different JavaScript virtual machine performance, with Opera consistently outperforming the rest, and Konqueror (the default KDE browser) coming last in performance. With ad-hoc computation you can't choose the computer the program is going to run eventually on, but it's always interesting to have this data at hand when trying to predict the performance of a particular problem, or estimate how much time a problem is going to take based on statistics of browser usage in a particular web site.

Performance also varies with the kind of problem (specifically depending on the kind of operations used to compute fitness, and the data types –integer or floating point–), so, in this occasion, a floating-point based problem has been chosen: the 10-variable Griewank [22] function,

$$F(x) = \sum_{i=1}^n \frac{x_i^2}{4000} + \prod_{i=1}^n \cos \frac{x_i}{\sqrt{i}} + 1 \quad (1)$$

$x_i$  is in the range -511,512. This function is characterized by a high number of local minima, although it can be easily solved using any global optimization procedure. We are not really interested in its difficulty, but in the fact that it has got a size and a complexity adequate to measure performance. In our experiments, we have chosen  $n = 10$ . The chromosome uses 20 bits to encode each floating point number, so that each *gene* is decoded by computing  $x_i = (M - m) \frac{c_i}{1048575} + m$ , where  $M$  and  $m$  are the range minimum and maximum, and 1048575 the biggest number

that can be coded with 20 bits and  $c_i$  is the binary value of the gene.

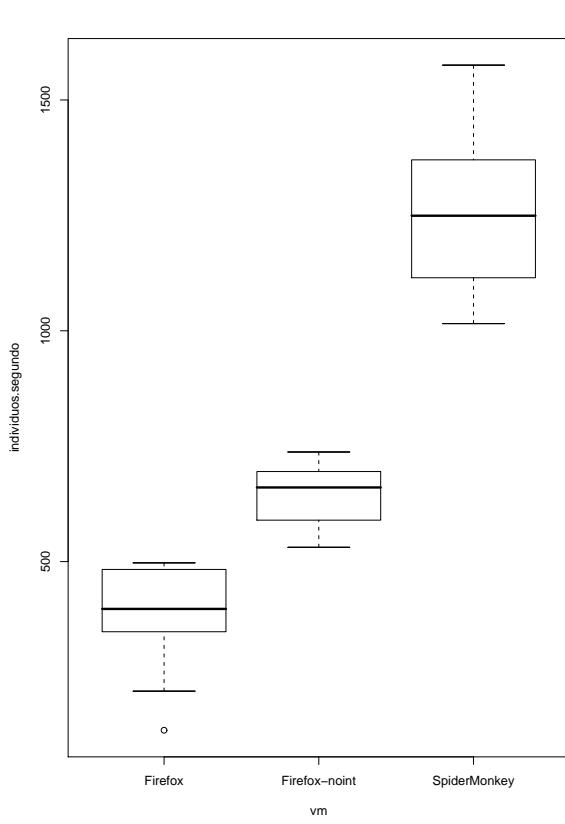
The experiments using this setup has been carried out as follows: first we have measured the individual evaluation rate in several configurations: a stand-alone Javascript interpreter (**JavaScript-C 1.5 2004-09-24** running on a Fedora Core 5 and a AMD Athlon(tm) 64 X2 Dual Core Processor 4200+), and Firefox<sup>10</sup> with two different settings: the default setting, and the *no-int*, which allows JavaScript programs to run without interruption for 30 seconds. The scripts were run several times (5 to 15) in the same machine with the usual, and similar, workload. Results in number of individuals evaluated per second are shown in figure 3.

These measures give us a baseline, without any intervention of the evolutionary algorithm, of a few hundred to one thousand chromosomes evaluated for fitness, per second, for this particular problem. Incidentally, it also indicates that the browser architecture and settings have a high impact on performance, which will have to be taken into account when trying to predict performance for a particular setting in advance<sup>11</sup>.

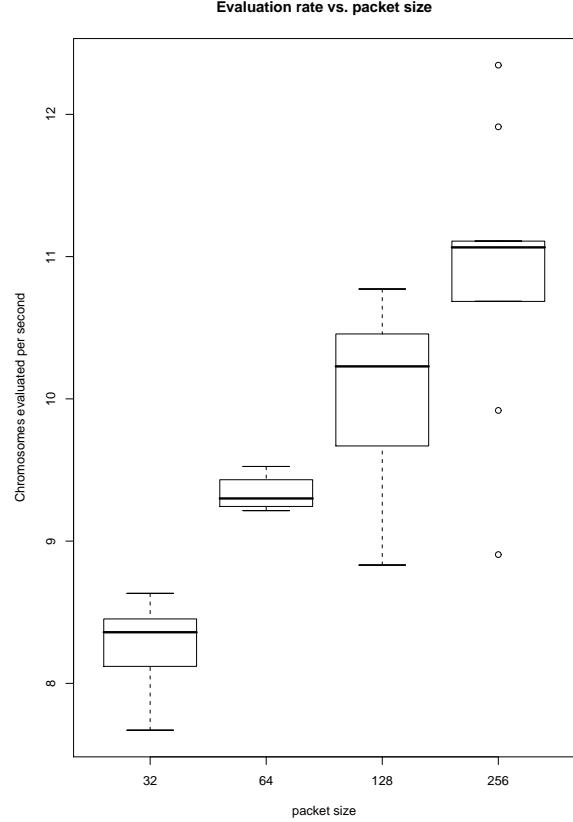
The second experiment will try to measure the impact of

<sup>10</sup>Mozilla/5.0 (X11; U; Linux x86\_64; es-ES; rv:1.8.0.8) Gecko/20061108 Fedora/1.5.0.8-1.fc5 Firefox/1.5.0.8

<sup>11</sup>Preliminary results with the Opera browser, not shown here, would be closer to the SpiderMonkey VM than to Firefox in either configuration



**Figure 3:** Boxplot of the number of evaluated chromosomes per second, for three different JS virtual machines: the SpiderMonkey stand-alone VM, Mozilla Firefox with script running time extended to 30 secs, and Firefox with running time limited to 10 secs (default setting).



**Figure 4:** Boxplot of the number of chromosomes whose fitness is evaluated in 1 second, depending on packet size in a client/server setup. Clearly, performance increases with packet size, which implies that latency and other overhead have a measurable impact on overall performance.

packet size on overall performance. Chromosomes are sent to the browser in packets of  $n$  individuals, which are then decoded, evaluated, and sent back to the server (just the ID and fitness). This takes some time, and generates overhead in the shape of database requests, data structure conversion, and the trip back and forth itself (latency). Initially, bandwidth is not an issue, at least from the client point of view, since these tests take place on a local area network (two computers connected to the same domestic ADSL router through an Ethernet 100Mbit/s connection).

Experimental setup is as follows: the server runs in the same computer as above, while the client runs in a Sony VAIO VGN-S4XP with an Intel Pentium M (2 GHz) running Firefox on Ubuntu 6.06, upgraded to the latest version (Jan 2007). An evolutionary algorithm that used 80% crossover and 20% mutation rate, a population of 512 with an elite (extracted for reproduction) of 256, and packet sizes of 32, 64, 128 and 256 was run several times. The number of evaluations was set to 5000, but, since the packet size is not a whole multiple of that amount, the simulation usually ended with a few more individuals evaluated. These were taken into account when computing the chromosome eval-

uation rate, shown in figure 4. This figure, which can be fitted by the lineal model  $n(s) = 8.36 \pm 0.19 * 0.011 \pm 0.001s$  with 99.9% confidence, shows that a new chromosome can be evaluated for every 100 that are added to the packet, and is obviously related to the number of petitions. A doubling of packet size slashes by half the number of request and responses from the server, decreasing also the number of database queries.

On the other hand, this model predicts that to achieve a performance similar to the figures shown above (500-1500 chromosomes/second) the packet size would have to be an unreasonable 90000; 100 chroms/second could be achieved with a packet sized around 8000. This hints at a way of squeezing more performance out of this setup (for instance, using a network with low latency, or increasing the speed of the database), but also points to a problem: bigger packet sizes means the client will be busy for more time, during which the client could just turn the computer off or wander away to another web page. In any case, it indicates that packet sizing will have to be considered carefully in browser-based distributed evolutionary computation, and also that the promises of massively parallel performance will only be

achieved when more than 20 computers are used to solve the problem. That is why it is interesting to see how the server, which is the bottleneck in scaling, behaves when the number of concurrent clients increases. This is what we are going to do next.

## 5. SCALING BEHAVIOR

In order to perform this experiment, an assortment of different computers, with speeds ranging from 750MHz to 2.8 GHz, were added, one by one. Packet size was set to 100 and equal to population size; the rest of the parameters are irrelevant. Connections also varied from a local connection in a 2-processor computer, to Fast Ethernet to WiFi. Experiment did not start at the same time in all computers, but more or less sequentially (actually, some people had to physically set the browser URL). Besides, very few of them were fully dedicated to the task; the URL was loaded while other people were working on the computer. That is why an improvement in averages should not really be expected, but we should expect, at least, an improvement in the best case, when all computers are started in a short period, and there is no net congestion or CPU overload in any of them. In every case, experiment was repeated several times.

That is what can be observed in figure 5, which shows the boxplot of the evaluation rate (total number of chromosomes evaluated divided by time in seconds, as measured from the information stored in the server log) vs. number of nodes. Scaling is dramatic from the first steps, but it lowers down when 4 or 5 computers are added. Best-case result always improves, but not dramatically, and average improvement slows down to a halt. This is due to a number of factors, not the least being that the last computer added was one of the slowest, but also to the fact that the server is not running in full production mode, and is spending some time logging information. There is also an architecture problem with the RoR server: while all requests run in its own thread, the request/response loop is sequential, so that just a single response can be served to the client simultaneously.

However, what is really important is the fact that we can obtain a best-case improvement, as we are showing here. If debugging is turned off (which we haven't done here for the full experiment since the information is needed to check that the algorithm is running correctly), some improvement can be obtained, as is shown in figure 6

The experiment shown in figure 6 was done using the same algorithm configuration as in 4, and an improvement of around 20% on average is obtained. This has an added advantage: logging messages are sequential, which means that threads handling a client must wait until other threads finish writing in that single file, provoking interlocking problems, which might also account for the poor average performance improvement observed above. We will try to fix this in the next version of the DCoR package.

## 6. CONCLUSIONS AND FUTURE WORK

The main purpose of this paper has been to introduce the DCoR (Distributed Computation on Rails) framework, make some measures to find out what kind of performance we can expect from it, and run scaling experiments on a simple configuration to highlight scaling problems with it. We conclude that, barring major optimization and tweaking of server performance, and using in each case the best

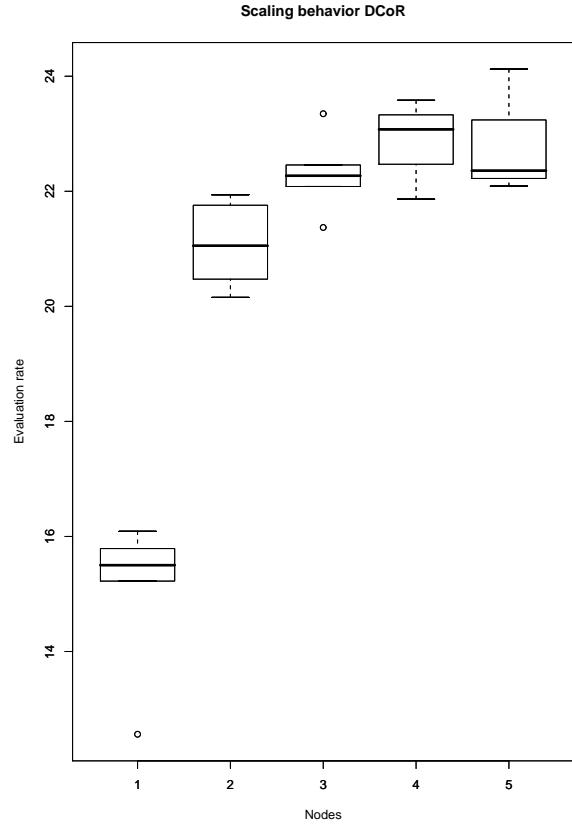


Figure 5: Boxplot of evaluation rate for a single server, and several clients, ranging from 1 to 5.

browser/client combination available, a good amount of clients is needed to equal the performance of a stand-alone machine running the same algorithm. But the problem is that, in the current setup, using a multi-threaded but single-process server, that amount of scaling cannot be achieved, with performance peaking when a few clients are added.

This leaves several possible paths for improvement: making DCoR fully reentrant, so that multiple copies can easily run at the same time in a server, and using a configuration of server clusters with a reverse proxy (which is not trivial, but not too difficult either), or changing the DCoR model so that more computation is moved to the clients, leaving the server as just a hub for information interchange among clients; that information interchange will have to be reduced to the minimum, and, if possible, a single chromosome per generation. That will make this model closer to the island model, being every browser running the experiment a more or less independent island, with just the migration policies regulated by the server. That way, the server bottleneck is almost eliminated. In the near future, we will try to pursue research along these two lines.

## Acknowledgments

This paper has been funded in part by the Spanish MICYT project NADEWeb: *Nuevos Algoritmos Distribuidos Evolutivos en la web*, code TIC2003-09481-C04. We also ac-

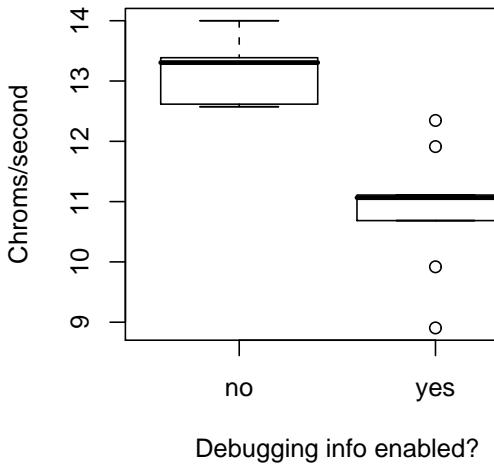


Figure 6: Boxplot of evaluation rate for a single server and client, in production mode and with debugging messages to log-file turned off (left), and in development mode with debugging messages on (right).

knowledge the support of the Spanish Rails (`ror-es`) mailing list.

## 7. REFERENCES

- [1] E. Alba. ¿Puede un algoritmo evolutivo paralelo proporcionar ganancia superlineal?, 1999. *Actas CAEPIA-TTIA'99* Vol. 2, pp. 89-97.
- [2] D. P. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.
- [3] M. Arenas, L. Foucart, J.-J. Merelo-Guervós, and P. A. Castillo. JEO: a framework for Evolving Objects in Java. In *Actas Jornadas de Paralelismo* [35], pages 185–191. <http://geneura.ugr.es/pub/papers/jornadas2001.pdf>.
- [4] A.-L. Barabási, V. W. Freeh, H. Jeong, and J. B. Brockman. Parasitic computing. *Nature*, 412(6850):894–897, August 2001.
- [5] C. Baray. Genetic algorithms as virtual user managers, November 1999. <http://cristobal.baray.com/indiana/GAVUM/>, last accessed Aug/2006.
- [6] P. Cappello and D. Mourloukos. A scalable, robust network for parallel computing. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 78–86, New York, NY, USA, 2001. ACM Press.
- [7] J. Castellano, P. Castillo, J.-J. Merelo-Guervós, and G. Romero. Paralelización de evolving objects library usando MPI. In *Actas Jornadas de Paralelismo* [35], pages 265–270.
- [8] J. G. Castellano, M. García-Arenas, P. A. Castillo, J. Carpio, M. Cillero, J.-J. Merelo-Guervós, A. Prieto, V. Rivas, and G. Romero. Objetos evolutivos paralelos. In U. d. G. Depto. ATC, editor, *XI Jornadas de Paralelismo*, pages 247–252, 2000.
- [9] P. Chong. Java based distributed genetic programming on the internet. In *Evolutionary computation and parallel processing Workshop, GECCO 99 conference*, 1999.
- [10] J. Culbert. Ajax and distributed computation thoughts. Published at <http://culbert.net/?p=6>, March 2006. Último acceso Mayo 2006.
- [11] ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, Dec. 1999.
- [12] ECMA. *ECMA-290: ECMAScript Components Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 1999.
- [13] ECMA. *ECMA-327: ECMAScript 3: Compact Profile*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, June 2001.
- [14] ECMA. *ECMA-357: ECMAScript for XML (E4X) Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 2004.
- [15] D. Flanagan. *JavaScript Pocket Reference (2nd Edition)*. O'Reilly, October 2002.
- [16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.
- [17] M. García-Arenas, J. G. Castellano, P. A. Castillo, J. Carpio, M. Cillero, J.-J. Merelo-Guervós, A. Prieto, V. Rivas, and G. Romero. Speedup measurements for a distributed evolutionary algorithm that uses Jini. In U. d. G. Depto. ATC, editor, *XI Jornadas de Paralelismo*, pages 241–246, 2000.
- [18] Gilorien. *DHTML and JavaScript*. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 2000.
- [19] G. A. Kohring. Implicit simulations using messaging protocols. *COMPUTERS AND PHYSICS*, 14:203, 2003.
- [20] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming (JOOP)*, 1(3):26–49, 1988.
- [21] R. M. Lerner. At the forge: assessing Ruby on Rails. *Linux J.*, 2006(142), February 2006.
- [22] M. Locatelli. A note on the Griewank test function. *J. of Global Optimization*, 25(2):169–174, 2003.
- [23] J. J. Merelo and J. Lupián. Computación distribuida sobre railes. In G. de Redes y Arquitecturas de Altas Prestaciones, editor, *Actas Jornadas de Paralelismo*, pages 367–372. Thomson, Septiembre 2006.

- [24] J. J. Merelo and J. Lupi  n. Escalado de computaci  n distribuida sobre rafles. In *Actas MAEB 2007*, 2007.
- [25] J.-J. Merelo-Guerv  s, J. Castellano, P. Castillo, and G. Romero. Algoritmos gen  ticos distribuidos usando SOAP. In *Actas Jornadas de Paralelismo* [35], pages 99–103.
- [26] V. M. Molina, J. J. M. Guerv  s, J. L. J. Laredo, and M. G. Arenas. Algoritmos evolutivos en Java: resoluci  n del TSP usando DREAM. In *Actas XVI Jornadas de Paralelismo, incluido en CEDI'2005. Granada, septiembre 2005*, pages 667–683. Thomson, Septiembre 2005.
- [27] D. S. Myers and M. P. Cummings. Necessity is the mother of invention: a simple grid computing system using commodity tools. *J. Parallel Distrib. Comput.*, 63(5):578–589, 2003.
- [28] Towards Open Grid Services Architecture. <http://www.globus.org/ogsa/>.
- [29] L. F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.
- [30] L. F. G. Sarmenta and S. Hirano. Bayanihan: building and studying Web-based volunteer computing systems using Java. *Future Generation Computer Systems*, 15(5-6):675–686, 1999.
- [31] C. Setzkorn and R. C. Paton. Javaspaces - an affordable technology for the simple implementation of reusable parallel evolutionary algorithms. In J. A. L  pez, E. Benfenati, and W. Dubitzky, editors, *Proceedings of the International Symposium on Knowledge Exploration in Life Science Informatics, KELSI 2004*, volume 3303 of *Lecture Notes in Artificial Intelligence*, pages 151–160, Milan, Italy, 25–26 Nov. 2004. Springer.
- [32] R. Shah. A beginner's guide to JavaScript. *JavaWorld: IDG's magazine for the Java community*, 1(1):??–??, Mar. 1996.
- [33] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, October 2004.
- [34] D. Thomas, D. Hansson, L. Breedt, M. Clark, T. Fuchs, and A. Schwarz. *Agile Web Development with Rails (The Facets of Ruby Series)*. Pragmatic Bookshelf, July 2005.
- [35] UPV. *Actas XII Jornadas de Paralelismo*. Universidad Pol  cnica de Valencia, 2001.
- [36] Wikipedia. Ajax (programming) — Wikipedia, the free encyclopedia, 2007. [Online; accessed 16-January-2007].